修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院 情報システム学研究科　　情報システム基盤学 専攻 博士前期課程		
氏　　名	片桐 国建	学籍番号	1153006
論 文 題 目	実行時情報を活用した第一級継続のオーバーヘッド削減手法		

要　旨

継続を第一級のオブジェクトとしてキャプチャする第一級継続という言語機能が知られている。この機能の用途は様々であるが、第一級継続機能は、例外処理を利用したプログラムへ変換することにより、言語処理系には手を入れることなく実現できることが知られている。JavaScript においては、長時間に渡る処理やブロックする処理によって他のイベント処理が停滞するのを避けるために第一級継続が利用できる場面がある。既に知られているプログラム変換では、ほとんどの関数呼び出しの部分に try-catch を挿入する。継続のキャプチャは例外処理の伝播を利用することで、通常ではアクセス不可能な関数のローカルな情報へアクセスし、必要な情報を保存することで実現する。

例外処理が含まれたプログラムは処理系の最適化を抑制する。近年、JavaScript の処理系の開発は活発に進んでおり多くの高度な最適化が行われている。しかし、第一級継続の実現のために例外処理が多く挿入されたプログラムは、この最適化が抑制され、例外処理の挿入に起因するオーバーヘッドが大きく増加する。JavaScript のような動的な言語では静的な解析によって try-catch 挿入を抑制することが困難である。

本研究ではこの問題を解決するため、実行時情報を活用し動的に try-catch を挿入／削除することでオーバーヘッドを削減する手法を提案する。また、本手法では投機的に try-catch の削除を行うため、大域脱出と再実行によって動的に try-catch の挿入を行い、try-catch が不足していた場合でも継続をキャプチャできる手法を提案している。また、1 つの関数に対して 2 種類の本体を持つように変換し、2 つの本体を呼び分けることで、動的な try-catch の挿入と削除を実現する。本研究においては、動的な try-catch の挿入と削除に関する戦略として、まず全ての関数呼び出しに try-catch を挿入した上で、実行時情報を元に順次 try-catch を削除していくことで高速化する戦略を示した。性能評価においては、第一級継続を用いたベンチマークプログラムによって提案手法が有効に機能するかを検証し、本研究の対象となるケースのプログラムにおいて性能向上することを確認した。

平成25年度 修士論文

実行時情報を活用した第一級継続のオーバーヘッド削減手法

電気通信大学 大学院情報システム学研究科

情報システム基盤学専攻

1153006 片桐国建

指導教官 小宮 常康 准教授
多田 好克 教授
大森 匡 教授

提出日 平成26年2月12日

目 次

図 目 次

第 1 章

はじめに

継続とは、プログラムの残りの計算を表すプログラミング言語共通の概念である。第一級継続とは、継続を概念としてではなく、第一級のオブジェクトとして取り扱えるようにする言語機能のことを指す。この言語機能は高度なプログラムの制御を可能にする機能であり、第一級継続を応用することで、大域脱出やコルーチンなどの制御機能を実現できる。また、JavaScript においては、長時間に渡る処理やブロックする処理によって他のイベント処理が停滞するのを避けるように記述するのが難しいという問題があるが、この問題の解決に第一級継続が利用できる場面がある。

第一級継続を JavaScript などの第一級継続をサポートしていない言語処理系で実現するためには、処理系を改造する方法が考えられる。しかし、処理系を改造する方法は言語の移植性を損なうという問題点がある。処理系に手を加えずに既存処理系において第一級継続を実現する手法としてプログラム変換が知られている。特に、比較的オーバーヘッドの少ない第一級継続実現のためのプログラム変換として例外処理を用いて追加する手法が知られている。

例外処理を利用した第一級継続実現のためのプログラム変換に関する従来研究として、Sekiguchi ら [4] 提案、Pettyjohn ら [6] の提案、Loitsch[5] らの提案があげられる。Sekiguchi ら [4] は Java のような命令型言語で部分継続を操作するための手法を紹介しており、オーバーヘッドの少ない継続操作の実現方法として例外処理を利用したプログラム変換を提案している。Pettyjohn ら [6] は第一級継続のキャプチャ実現のための理論的なモデルと、ランタイムスタックへのアクセスをサポートしていない場合でも第一級継続を実現するための変換を MSIL.Net の仮想マシン

を例にあげて紹介している。Loitsch[5] は、JavaScript における継続のキャプチャを実現するための例外処理を用いたプログラム変換を提案している。

従来研究として知られている第一級継続実現のためのプログラム変換では、プログラム中に多くの例外処理コードを挿入する。しかし、例外処理コードは処理系の最適化を抑制することが知られている。近年、JavaScript の処理系の開発は活発に進んでおり多くの高度な最適化が行われているが、第一級継続の実現のために例外処理が多く挿入されたプログラムは、この最適化が抑制され、例外処理の挿入に起因するオーバーヘッドが大きく増加する。

本研究では，例外処理コードを削減し、例外処理コードの挿入に起因するオーバーヘッドを軽減するため手法を提案する。例外処理コードを削減するために動的に例外処理コードの削除／挿入するためしくみを実現する。

本稿の内容を章ごとに紹介する。第 2 章では、研究の背景として、第一級継続の概要、既存の第一級継続の実装手法を紹介し、また既存手法の問題点を指摘する。第 3 章では、既存手法である例外処理を用いた継続のキャプチャ手法を実現する変換を紹介する。第 4 章では、提案手法である動的な try-catch 挿入の実現の手法を説明する。第 5 章では、動的な try-catch 挿入を実現するプログラム変換について述べる。第 6 章では、2 つのベンチマークプログラムに対して、既存手法でのオーバーヘッドと提案手法でのオーバーヘッドの計測を行い、結果を比較する。

第 2 章

背景

2.1 第一級継続

継続とは、プログラムの実行において、ある特定の場所以降でやり残されている計算を表す概念である。「第一級」とは、そのプログラミング言語における第一級のオブジェクトとして扱えることを指す。第一級のオブジェクトとは**変数への直接代入**や**関数の引数や返り値として使用**できるオブジェクトのことである。第一級継続とは、変数への直接代入や関数の引数や返り値として使用できる継続オブジェクトを実現する機能を指す。

プログラミング言語 Scheme[1][2] において、call-with-current-continuation (call/cc) というプリミティブ関数がある。この関数を使うことで、継続を関数オブジェクトとして取り出すことができる。コード 2.1 に call/cc を使ったシンプルなプログラムを実行した例を示す。

ソースコード 2.1: call-with-current-continuation の使用例

```
1  (define cont '())
2  (begin
3   (display 'a)
4   (call/cc
5    (lambda (c)
6      (set! cont c)
7      (display 'b)))
8   (display 'c)) ;; => abc
9  (cont) ;; => c
10 (cont) ;; => c
```

call/cc は、call/cc 実行後の継続を関数として取り出す。コード 2.1 において call/cc 実行後の継続は (diaplay 'c) である。グローバル変数である cont には、(diaplay 'c)

を実行する内容の継続が保存される。以降、継続 cont を呼び出すたびに (diaplay 'c) という内容が実行される。

2.1.1 第一級継続の応用例

call/cc を使って非局所的脱出 (non-local exit) やコルーチン (co-routine) などの高度な制御機能を実現することができる。図 2.2 に call/cc によって非局所的脱出を実現したコードの例を示す。

ソースコード 2.2: call-with-current-continuation の使用例

```
1 (call/cc
2  (lambda (break)
3    (let loop ((x 5))
4      (if (< x 0) (break "break"))
5      (display x)
6      (newline)
7      (loop (- x 1)))))
```

4 行目の条件を満たした時に break 関数を実行することで、C 言語の longjump のように制御を call/cc の外側に移すことができる。

2.1.2 JavaScript における第一級継続の利用場面

現在、Web アプリケーションの開発において Ajax 技術が一般的に用いられている。これにより、クライアントサイドで動作する複雑なプログラムが書かれるようになった。

JavaScript はシングルスレッド上で動作するプログラミング言語である。JavaScript では非同期処理をイベント駆動型でしか記述できないため、プログラムが分断され、制御フローの分かりやすい記述が困難となる。

非同期のコードを読みやすく記述するための機能として、第一級継続、スレッド、ファイバー、コルーチンなどがあげられる。このような機能は一部の JavaScript 実装では使えることがあるが、ほとんどの環境では利用することはできない。

2.2 第一級継続の実装

高速な第一級継続の実装の戦略に関する Clinger[3] の報告がある。実行中の関数のリターンアドレス，ローカル変数などを保存しておくデータ構造をランタイムスタックと呼び、第一級継続の実現のためにはランタイムスタックへのアクセスが必要である。処理系を改造することで、ランタイムスタックへのアクセスをサポートし、第一級継続を実現することができる。しかし、この方法は移植性を損なうという問題がある。

2.3 例外処理を活用したプログラム変換による第一級継続の実現

例外処理を利用した第一級継続実現のためのプログラム変換においては移植性の問題はない。この節では、既存手法としてそのプログラム変換を説明する。継続の操作を行う機能の実現には、ランタイムスタックが保持する情報にアクセスする必要がある。既存手法では、ランタイムスタックへのアクセスに相当する処理を実現するために例外処理を用いている。既存手法では、関数呼び出し部に try-catch を挿入するプログラム変換を行う (図 2.1 参照)。図 2.1 はプログラムの実行の流れを表した図であり、縦の矢印は関数の内容、右斜め矢印の矢印は関数呼び出し、黒丸は例外処理の挿入箇所を表している。suspend は継続をキャプチャするための関数である。継続をキャプチャを行う際に図 2.1 にあるように例外を throw、例外の伝播を行うことで、呼び出し元の全ての関数ローカルな情報にアクセスし、ローカル変数の情報や、その関数における残りの計算に相当するデータ (残りの計算を表すクロージャオブジェクト) を保存していく。

コード 2.3 に例外処理を用いたプログラム変換後の fibonacci 関数のコードを示す。継続のキャプチャを行う際に関数ローカルな情報にアクセスするために、全ての関数呼び出し try-catch が挿入されていることが確認できる。継続のキャプチャの際に保存する呼び出された各関数のコードの残りの計算は、関数呼び出しごと

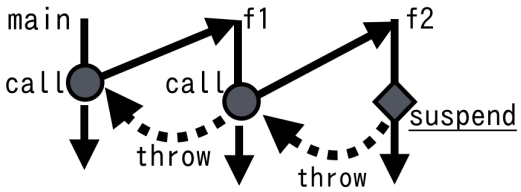

図 2.1: 例外の伝播による継続のキャプチャ

に分割された関数 (k1〜k3) として定義されている。catch 節では、伝播してきた例外オブジェクトに対して残りの計算に相当する関数を保存するためのコードを追加している。

ソースコード 2.3: 例外処理を利用したプログラム変換のコード例

```
1  // 変換前
2  var fib = function( x ){
3    return ( x <= 1 ) ? x : fib( x − 1 ) + fib( x − 2 );
4  };
5  // 変換後
6  var fib = function ( x ) {
7    var t1, t2, t3, t4, t5, t6;
8    var k1 = function() {
9      t1 = x <= 1.0;
10     if (t1) {
11       return x;
12     }
13     t2 = x − 1.0;
14     try {
15       t3 = fib( t2 );
16     } catch ( ex ) {
17       if ( ex instanceof ContinationException ) {
18         ex.push(function ( t3 ) { // 残りの計算 (継続)に相当する関数
19           return k2( t3 );
20         });
21       }
22       throw ex;
23     }
24     return k2(t3);
25   };
26   var k2 = function( t3 ){
27     t4 = x − 2.0;
28     try {
29       t5 = fib( t4 );
30     } catch ( ex ) {
31       if ( ex instanceof ContinationException ) {
32         ex.push(function( t5, t3 ){ // 残りの計算 (継続)に相当する関数
33           return k3( t5, t3 );
34         });
35       }
36       throw ex;
37     }
38     return k3( t5, t3 );
39   };
```

```
40    var k3 = function( t5, t3 ){
41      t6 = t3 + t5;
42      return t6;
43    };
44    return k1();
45  };
```

2.3.1 例外処理の追加によるオーバーヘッド増加の問題

例外処理コードは処理系の最適化を抑制することが知られている。近年、JavaScript の処理系の開発は活発に進んでおり多くの高度な最適化が行われているが、第一級継続の実現のために例外処理が多く挿入されたプログラムは、この最適化が抑制され、例外処理の挿入に起因するオーバーヘッドが大きく増加する。

既存手法では原則、全ての関数に try-catch を挿入する。しかし、try-catch の挿入は最適化の妨げになり、オーバーヘッドを大きく増加させる。また、全ての関数呼び出しのうち、実際に try-catch 挿入が必要なのは一部の関数呼出しだけである場合もある。例えば、図 2.2 のような場合では 6 つの関数呼び出しがあるが、実際に継続のキャプチャ時に例外の伝播が必要となる関数の呼び出しは f1、g の呼び出しの 2 つだけである。

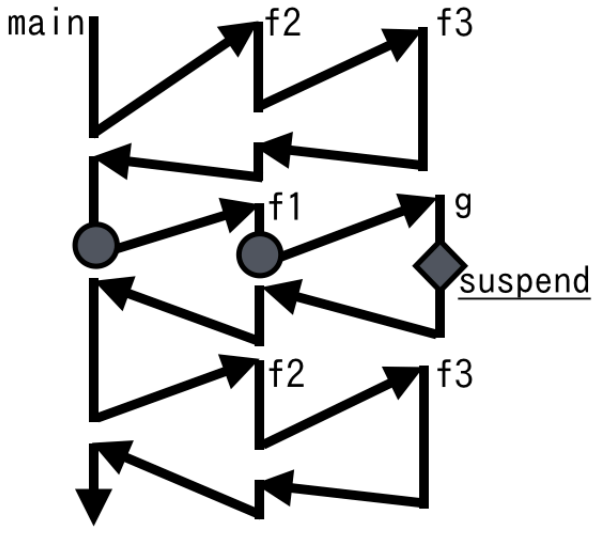

図 2.2: 実際に継続キャプチャに必要される try-catch 挿入部分

2.3.2 静的プログラム解析の限界

既存手法で、try-catch の挿入を抑制する方法として静的プログラム解析が考えられる。静的プログラム解析では、実際にプログラムを実行することなくプログラムを解析する。しかし、JavaScript は動的なプログラミング言語であり、高階関数、プロトタイプベースのオブジェクトシステムなどをサポートする。このようなプログラミング言語は、実際に実行するまで、どの関数が呼び出されるか分からない場合が多く存在する。JavaScript では静的プログラム解析によって正確な情報を得ることが難しい。

第 3 章

例外処理を用いた第一級継続の実現

既存の継続の実装手法として知られている例外処理を用いたプログラム変換 [4][6][5] の詳細を紹介する。この手法は本研究の手法のベースとなっている。

3.1 継続のキャプチャ(suspend 関数)

既存手法で継続のキャプチャをするための関数として suspend 関数を紹介する。コード 3.3 の 16 行目で用いられている suspend 関数は、継続のキャプチャをする関数である。suspend 関数は以下のような 1 引数の関数として用意されている。

```
suspend( <関数> )
```

suspend 関数は一つの関数を引数として受け取る。suspend 関数は実行されると、現在の継続 (current continuation) をキャプチャし、その継続オブジェクトを引数の < 関数 > に渡し、その < 関数 > を実行する。suspend 関数は引数の関数を実行し終えると、そのときの処理を終了する。コード 3.1 に suspend 関数を用いた sleep 関数のプログラムの実装例を示す。

ソースコード 3.1: suspend 関数の使用例 (sleep)

```
1 function sleep( ms ) {
2   suspend( function( cont ){
3     setTimeout( cont, ms );
4   } );
5 };
```

コード 3.2 に suspend 関数の実装例を示す。

ソースコード 3.2: suspend 関数

```
1 function suspend( func ) {
2   var cex = new ContinuationException();
3   cex.setFunc( func );
4   throw cex;
5 }
```

suspend 関数は継続キャプチャのための例外を throw し、例外の伝播によって全ての呼び出し元の関数ローカルな情報にアクセスし、継続オブジェクトを作成すると、渡された関数に継続オブジェクトを渡して関数を実行する関数である (図 3.1 参照)。具体的には suspend 関数は内部で、継続例外オブジェクトを生成し (2 行目)、渡された関数を例外オブジェクトに関連付け (3 行目)、その例外オブジェクトを throw する (4 行目)。

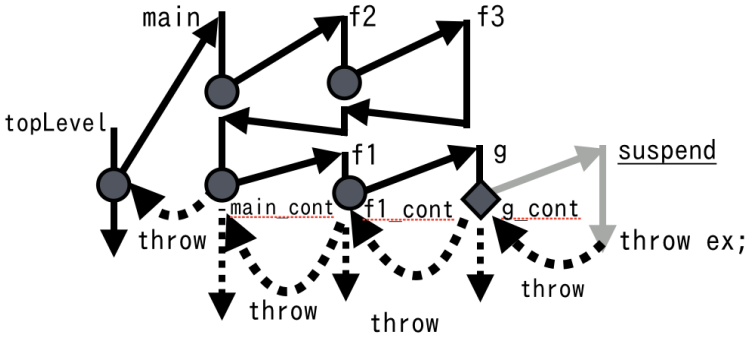

図 3.1: 継続のキャプチャ

3.2 関数呼び出しへの try-catch 挿入

既存手法で継続キャプチャを行う際には、例外処理の伝播を利用し、呼び出し元の全ての関数毎のローカルな情報にアクセスする。コード 3.3 を例に、継続のキャプチャを実現するプログラム変換を説明する。

ソースコード 3.3: 既存手法の説明に用いるサンプル

```
1 var f1 = function( x, g ) {
2   print('f1:␣' + x);
3   return g();
4 };
```

まずは、A 正規化と呼ばれる変換を行う。この変換は**計算の途中経過に対して変数を明示的に割り当てる変換**である。コード 3.4 に A 正規化後の f1 のコードを示す。

ソースコード 3.4: A 正規化後のコード

```
1 var f1 = function (x, g) {
2   var t1, t2;
3   t1 = "f1: " + x;
4   print(t1); // safeCall
5   t2 = g(); // unsafeCall
6   return t2;
7 };
```

A 正規化されたことで関数呼び出し毎に try-catch が追加することが可能となった。try-catch を追加した結果をコード 3.5 に示す。

ソースコード 3.5: try-catch を追加する変換後のコード

```
1  var f1 = function ( x, g ) {
2    var t1, t2;
3    var k1 = function() {
4      t1 = "f1: " + x;
5      print( t1 );
6      try {
7        t2 = g();
8      } catch ( ex ) {
9        if ( ex instanceof ContinuationException ) {
10         ex.pushFrame(function( t2 ){
11           k2( t2 );
12         });
13       }
14       throw ex;
15     }
16     return k2( t2 );
17   };
18   var k2 = function( t2 ) {
19     return t2;
20   };
21   return k1();
22 };
```

関数 g の呼び出し (7 行目) を try ブロック内におさめており、また catch 節では継続キャプチャの際に throw される特別な例外が備えた条件分岐を用意している。継続キャプチャをする際には、図 2.1 のように throw を繰り返して、呼び出し元へ、呼び出し元へと遡っていく。

例外の伝播を利用して、呼び出し元へ遡る際にそれぞれの関数ごとの残りの計算を関数として保存し、継続オブジェクトに追加していく (図 3.1 参照)。継続オブ

ジェクトの中身は遡る際に通った各関数の残りの計算を表す関数オブジェクトを格納した配列である。この配列の関数オブジェクトを順番に実行することで保存された継続を実行することができる。

第 4 章

動的な try-catch の追加・削除

既存手法ではほとんどの関数呼び出しに try-catch を追加せざるをえないことが、処理系の最適化の妨げとなっており、オーバーヘッド増加の大きな原因となっている。本研究では、この問題を解決するために、動的に try-catch 挿入／削除を決定するオーバーヘッド削減手法を提案する。本章では、動的な try-catch 挿入／削除の実現方法と、どのように 2 種類の関数本体を呼び分けるかの戦略を説明する。

4.1 関数呼び出しにおける try-catch の要／不要の動的な切り替え

本手法では、各関数の定義について、関数本体に現れる全ての関数呼び出しに try-catch を挿入した版と try-catch を挿入しない版の 2 種類の本体を持つように定義した関数を作成するプログラム変換を行う。また、関数には、現在の関数が 2 種類の本体のどちらを呼び出すか、過去に継続キャプチャに利用されたかの真偽、呼び出し回数の情報を保持する状態変数を持たせる。状態変数は外部アクセス可能なメソッドを実行することで取得、書き換えが可能である。各関数は、状態変数の値によって 2 種類の関数の本体のどちらを呼び出すかを決定する。

ソースコード 4.1: 提案手法によるプログラム変換の例

```
1 // 変換前
2 var f = function ( i ) {
3   if( i < 1 ) return;
4   f( i − 1 );
5 };
6 // 変換後
7 var f = (function(){
```

```
8    // try−catch を含まない関数
9    var funcA = function ( i ) {
10     if( i < 1 ) return;
11     f( i − 1 );
12   };
13   // try−catch を含む関数
14   var funcB = function ( i ) {
15     if ( i < 1 ) return;
16     t1 = i − 1.0;
17     try { f( t1 ); } catch( ex ) {
18       /* 継続キャプチャのためのコード */
19     }
20   };
21   var currentFunc = funcB;
22   var self = function( i ){ currentFunc( i ); };
23   /* 外部から関数の内部状態を変更するためのコード */
24   return self;
25 })();
```

変換された関数の状態は 3 つの状態のいずれかになる。

- try-ctach を含まない呼び出しを行う状態 (1)
- try-ctach を含む呼び出しを行う状態 (2)
- 継続のキャプチャに利用され、必ず try-ctach を含む呼び出しを行う関数 (3)

図 4.1 に関数の状態がどのように遷移するかを示す。

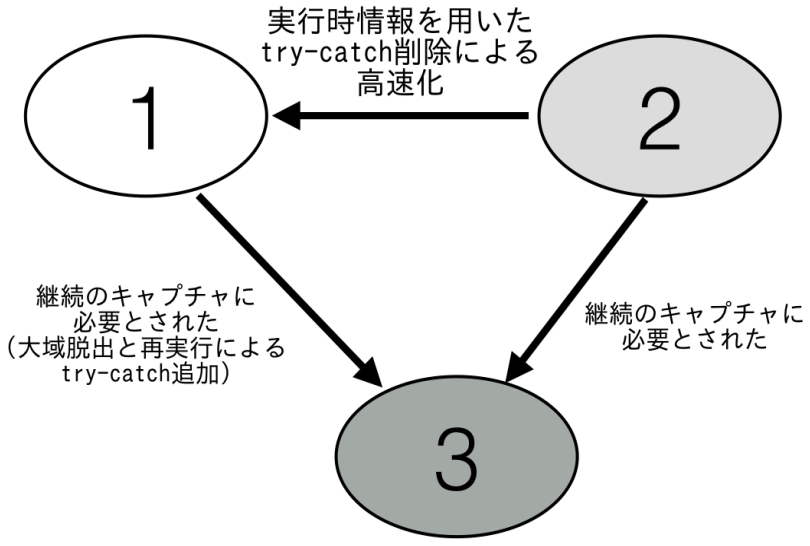

図 4.1: 関数の状態の遷移

初期状態では、ほとんどの関数が (2) の状態である。テスト実行によって、実行時情報を用いた try-catch の削除を行うことで、継続のキャプチャに用いられる可能性が低いと判断された関数は try-catch が削除され (1) の状態に移行する。(1) の状態の関数がもし継続のキャプチャ時の例外の伝播のパス上になり、関数ローカルの情報が必要とされた場合には、大域脱出を行い、(1) の状態の関数に try-catch の挿入を行い (3) の状態に変更した上で再実行することで対処する。(2) の状態においても、一度継続のキャプチャに用いられた関数は (3) の状態に移行し、以降は動的な try-catch 削除の対象とならない。

4.2 実行時情報を活用した try-catch 削除

本手法では、実行時情報を活用し動的に try-catch を削除することでオーバーヘッドを削減する。実行時情報をプロファイラによって監視し、継続のキャプチャに利用される可能性が低い関数を検出し、try-catch を削除する。

4.2.1 プロファイラ

プロファイラは実行時情報として特に関数の呼び出し回数を監視する。プロファイラは、任意の関数の呼び出し回数が事前に定めた値を超えたたことを検出すると、関数の try-catch を削除する。またプロファイラは継続キャプチャに一度でも利用された関数の監視は行わない。

プロファイラは投機的に try-catch の削除を行う。投機的に try-catch の削除を行う場合、try-catch を削除してしまった関数でも、継続キャプチャに必要とされる可能性がでてきてしまう。継続のキャプチャ時の try-catch が不足している場合 (図 4.3 参照)、そのままでは継続のキャプチャは失敗し、正常なプログラムの続行が不可能となる。try-catch が不足にしている場合でも、継続のキャプチャを行い正常に処理を続けるためのしくみが必要である。

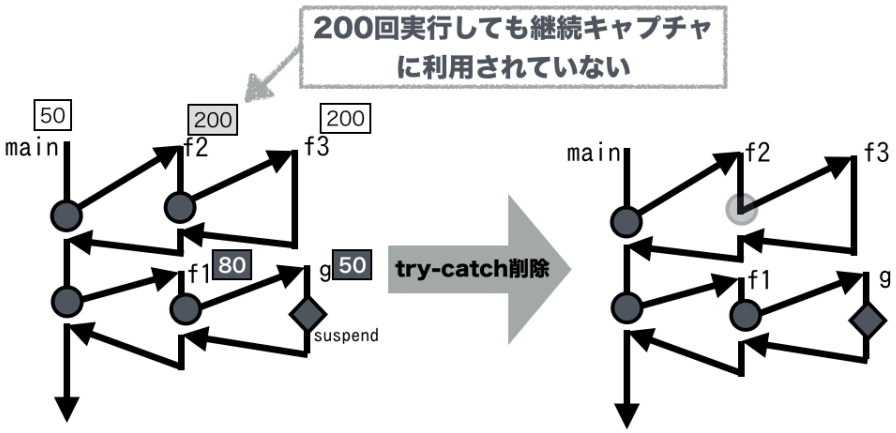

図 4.2: 関数の呼び出し回数を利用した try-catch の削除

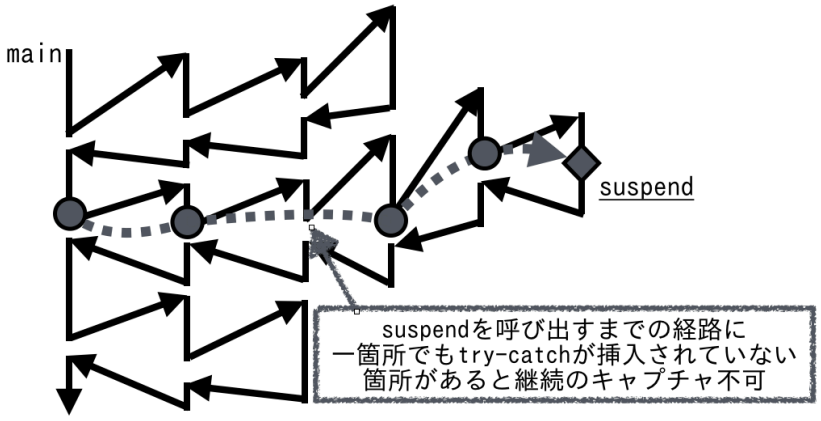

図 4.3: try-catch 挿入の不足

4.2.2 大域脱出と再実行による動的な try-catch 挿入

そこで、try-catch の不足時に継続のキャプチャを行うために、大域脱出によって実行を戻し、動的な try-catch 挿入をした上で再実行を行う手法を提案する。try-catch の不足時に継続のキャプチャを行う場合には、一度大域脱出を行い、決められた再実行ポイントまで制御を移す。継続キャプチャを行う関数までの経路上全ての関数に try-catch を動的に挿入しながら再実行を行う。再実行により、再び継続キャプチャの関数まで辿り着いたときには必要な箇所に try-catch が挿入されており、既存手法と同じように継続キャプチャを行う (図 4.4 参照)。

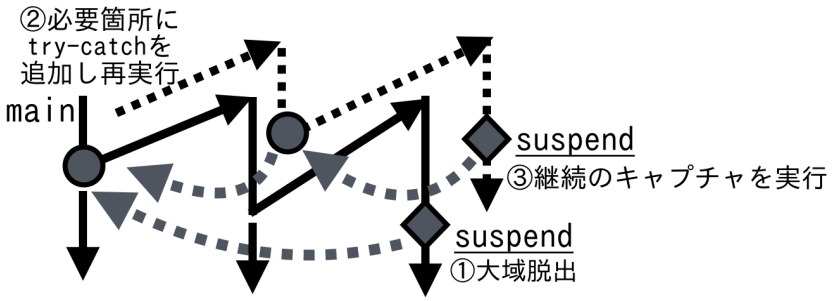

図 4.4: 動的な try-catch 挿入

一度、継続のキャプチャの例外の伝播の経路上になった関数は、その情報が関数の状態変数に保存され、以降は実行時情報を用いた try-catch 削除の対象にならない。

4.2.3 再実行時の副作用の問題

図 4.4 にあるように try-catch の挿入不足時には大域脱出を行い、プログラムを再実行することで動的な try-catch 挿入を実現している。このとき、再実行されるプログラム中に**環境を書き換えるような命令**、**ネットワーク通信**などの副作用のある命令が含まれる可能性がある。これらの副作用を持つ命令を、再実行時にそのまま実行することは許されないため、何らかの対処をする必要がある。

この問題のうち**環境を書き換えるような命令**に関しては、自由変数を削除する変換が知られている。康 [7] の論文では、メモ化適用範囲の拡張のために自由変数を除去するラムダリフティングという変換手法を採用しており、本研究においても同様の変換手法を用いることで、環境を書き換えるような命令を除去することが可能である。**ネットワーク通信**に関しては、ネットワーク通信を管理するラッパーライブラリの作成を行い、プログラム変換時にはネットワーク通信をラッパー経由で行うよう変換することで対処する。ラッパーにおいて再実行時に 2 回めの命令が実行されないようにすることで対処がある程度可能だと考えられる。

4.3 本手法導入における分岐の増加と try-catch 削減のトレードオフ

本手法では、従来の手法における try-catch 挿入に加えて、実行時情報を収集するコード、動的に try-catch 有無を切り替えを可能にする条件分岐のコード、try-catch の不足時に try-catch を挿入し関数呼び出しを再実行するためのコードが追加される。try-catch の有無に関わらず、関数内部にいくつかの分岐が追加されるため、これはオーバーヘッド増加の原因となる。提案手法導入によるオーバーヘッド増加と try-catch の挿入削減によるオーバーヘッド削減はトレードオフの関係となり、一定以上の try-catch の削減ができなければオーバーヘッドが逆に増加する危険がある。

第 5 章

動的なtry-catchの追加・削除のためのプログラム変換

コード 5.1 を例に提案手法によるプログラム変換を紹介する。コード 5.1 を提案手法によって変換した結果としてコード 5.2 を示す。

ソースコード 5.1: 提案手法の説明に使う適当な関数

```
1  var f1 = function( x, g ) {
2    print('f1:␣' + x);
3    return g();
4  };
```

ソースコード 5.2: 提案手法によるプログラム変換例

```
1  var f1 = (function(){
2    // status
3    var __state = FUNC_B; // 状態
4    var __call_count = 0; // 関数f1 の呼び出された回数
5    var __is_access = false; // 状態
6    // low cost
7    var funcA = function f1( x, g ) {
8      print('f1:␣' + x);
9      return g();
10   };
11   // high cost
12   var funcB = function f1( x, g ) {
13     /* try−cacth が挿入された関数の本体のコード */
14   };
15   // return function
16   var self = function( x, g ) {
17     ++__call_count;
18     // キャプチャ失敗時の再実行
19     if ( FailCapture.flg ) {
20       __state = FUNC_B;
21       return funcB( x, g );;
22     }
23     return self.body(x, y, z);
24   };
25   // self.body はデフォルトで try−catch 有りの関数(2の状態)
```

```
26    self.body = function( x, g ){
27      var result;
28      // FUNC_B call
29      if ( CallObserver.isPassWhite() )
30        CallObserver.inc();
31      result = funcB( x, g );
32      if ( CallObserver.isPassWhite() )
33        CallObserver.dec();
34      return result;
35    };
36    // check
37    self.check = function(){
38      if (__call_count > T_FUNCAL_COUNT &&
39          !__is_access) {
40        self.chageFunc(FUNC_A);
41        return true;
42      }
43      return false;
44    };
45    // chage
46    self.chageFunc = function ( state ) {
47      /* try−catch 有無を変更するコード */
48    };
49    self.accessState = function(){
50      return __state;
51    };
52    self.info = function() {
53      return {
54        state: __state,
55        call_count: __call_count,
56        is_access: __is_access
57      };
58    };
59    self.fname = "f1";
60    Profiler.registFunction( self );
61    return self;
62  })();
```

本章では、コード 5.2 を例に提案手法によるプログラム変換を説明する。まず、コード 5.2 の概要を述べる。コード 5.2 の内訳は以下のようになる。

- 3 – 6 行 関数の状態を管理する変数宣言
- 7 – 10 行 try-catch を含まない f1 の関数の本体 (funcA)
- 12 – 14 行 try-catch を含む f1 の関数の本体 (funcB)
- 16 – 24 行 再実行時に動的に try-catch を追加するための分岐を行うために関数の本体をラップしている関数
- 26 – 35 行 実行時情報を用いた高速化の際に funcB から funcA に書き換えるための変数として self.body を用意している。デフォルトでは、funcB がセットされている。また継続のキャプチャに使われた場合には funcA から

funcB に切り替わる。

- 37 – 44 行 実行時情報 (呼び出し回数) をチェックし、それに応じて関数の状態を切り替えるための外部アクセス可能な関数
- 46 – 48 行 関数の状態を切り替えるための外部アクセス可能な関数
- 49 – 51 行 関数の状態を取得するためのの外部アクセス可能な関数
- 60 行 プロファイラへ、自身の参照を登録する命令

本手法では、既存手法に try-catch の挿入を動的に行う機構を追加し、オーバーヘッドの削減を図っている。本手法により変換された関数は 4.1 節で述べたいずれかの状態を持つ。

5.1 関数の状態の管理

コード 5.2 の 3 – 6 行の__から始まる変数によって関数は状態を管理される。3 行目で宣言されている変数*__state* には、その時の try-catch の要／不要を決める値が格納される。4 行目で宣言されている変数*__call_count* には、関数の呼び出し回数が保存される。5 行目で宣言されている変数*__is_access* には、その関数が継続のキャプチャに一度でも利用されたかを判別する真偽値が格納される。26 行目の self.body は呼び出される関数の本体である。デフォルトでは try-catch を含む関数 (funcB) を呼び出す関数が格納されており、状況に応じて try-catch を含まない f1 の関数の本体 (funcA) か try-catch を含む f1 の関数の本体 (funcB) のどちらかに書き換えられる。また、これらの状態は 37 – 51 行の外部アクセス可能な関数によって、値の取得、書き換えが可能である。

図 5.1 は変数によって関数がどのような状態に移行するかを示している。変数*__state* が FUNC_B でかつ変数*__is_access* が false のときに関数の状態は右上の状態となる。変数*__state* が FUNC_A でかつ変数*__is_access* が false のときに関数の状態は左上の状態となる。変数*__is_access* が true のときには下の状態となる。関数の呼び出し回数*__call_count* の値は直接的には図 5.1 の関数の状態の決定には使われないが、後述するプロファイラによって、変数*__state* を変更する際の判断に

使われる。

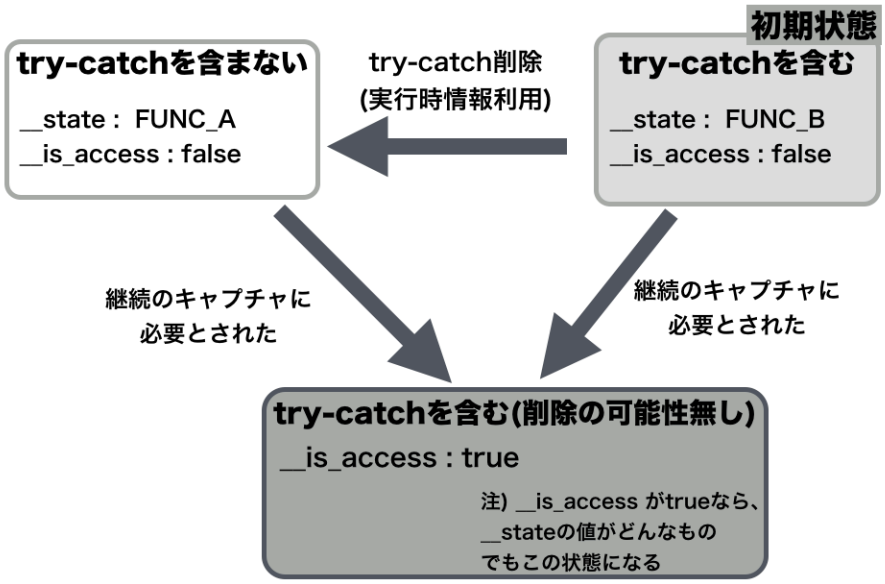

図 5.1: 状態変数による関数の状態の管理

5.2 二種類の関数の本体と呼び分け

self.body は実際に呼び出される関数の本体であり、コード 5.2 の 37 – 44 行の self.check 関数と、46 – 48 行の self.chageFunc 関数は関数の状態を変更する関数である。これらの関数は、self.body の中身と__state 変数を変更する。

self.check 関数は実行回数があらかじめ定めたしきい値を超えていて、かつ過去に継続キャプチャに使用されたかどうかを判定する。もし、条件を満たすならば、“self.chageFunc(FUNC_A)” が実行され、self.body は funcA に書き換えられ、__state 変数は FUNC_A に書き換えられる。関数 f1 はこれ以降、try-catch を含まない関数呼び出しを行う。

self.chageFunc 関数は関数の状態を変更する関数であり、try-catch の要／不要を決める変数*__state* と self.body の中身を変更する。コード 5.3 に関数 self.chageFunc の内容を示す。

ソースコード 5.3: 2 種類の本体を切り替える関数

```
1  self.chageFunc = function ( state ) {
2    if ( state ) {
3      __state = FUNC_A;
4      self.body = function( x, g ) {
5        var result;
6        ++__call_count;
7        CallObserver.passWhite();
8        result = funcA(x, y, z);
9        if ( CallObserver.black_count() === 0 )
10         CallObserver.reset();
11       return result;
12     };
13   } else {
14     __state = FUNC_B;
15     self.body = function( x, g ) {
16       var result;
17       ++__call_count;
18       if ( CallObserver.isPassWhite() )
19         CallObserver.inc();
20       result = funcB( x, g );
21       if ( CallObserver.isPassWhite() )
22         CallObserver.dec();
23       return result;
24     };
25   }
26 };
```

関数 self.chageFunc は引数 state の値が FUNC_A のとき、コード 5.3 の 3〜12 行にあるように、変数__state の値を FUNC_A に書き換え、self.body を funcA が呼び出される関数に書き換える。関数 self.chageFunc は引数 state の値が FUNC_B のとき、コード 5.3 の 14〜23 行にあるように、変数__state の値を FUNC_B に書き換え、self.body を funcB が呼び出される関数に書き換える。このとき、書き換えられる関数には関数の呼び出しカウント命令や、CallObserver.isPassWhite 関数を用いた条件分岐、CallObserver.inc 関数や、CallObserver.dec 関数の呼び出し呼び出しなどの命令が追加されている。これは「大域脱出・再実行の実現」のためのコードであり、後述する。

5.3 大域脱出・再実行の実現

提案手法では図 4.4 にあるように try-catch の挿入不足時には大域脱出を行う。このためには、実行中の関数のすべてに try-catch の挿入がされているかの判定と、大域脱出時にどこまで呼び出しを遡るかを決定する仕組みを必要とする。実行中

の関数において、try-catch が不足しているかを常に監視するグローバルオブジェクトとして、CallObserver オブジェクトを用意した。CallObserver の内容をコード 5.4 に示す。

ソースコード 5.4: 実行中の関数を監視するオブジェクト

```
1  var CallObserver = (function() {
2    var black_count = 0;
3    var pass_through_white = false;
4    return {
5      inc: function() {
6        ++black_count;
7      },
8      dec: function() {
9        −−black_count;
10       if ( black_count === 0 )
11         pass_through_white = false;
12       if ( black_count < 0)
13         throw 'black_exception';
14     },
15     passWhite: function() {
16       pass_through_white = true;
17     },
18     reset: function() {
19       pass_through_white = false;
20       black_count = 0;
21     },
22     isPassWhite: function() {
23       return pass_through_white;
24     },
25     black_count: function() {
26       return black_count;
27     }
28   };
29 })();
```

CallObserver オブジェクトは、内部に実行中の関数を監視し、現在継続キャプチャが可能か不可能かの情報 (変数 *pass_through_white*) と、継続キャプチャが不可能な場合には、try-catch を含まない関数からいくつの try-catch を含む関数呼び出しを経由したかの情報 (変数 *black_count*) を持つ。これらの情報は try-catch の不足の監視と再実行開始ポイントの決定に使われる。

5.3.1 try-catch の不足の監視と再実行開始ポイントの決定

try-catch の不足の監視と再実行開始ポイントの決定を実現するしくみを図 5.3 に示す。また、このしくみを実現するために CallObserber オブジェクトがどのよう

に動作するかを図 5.3 を示す。

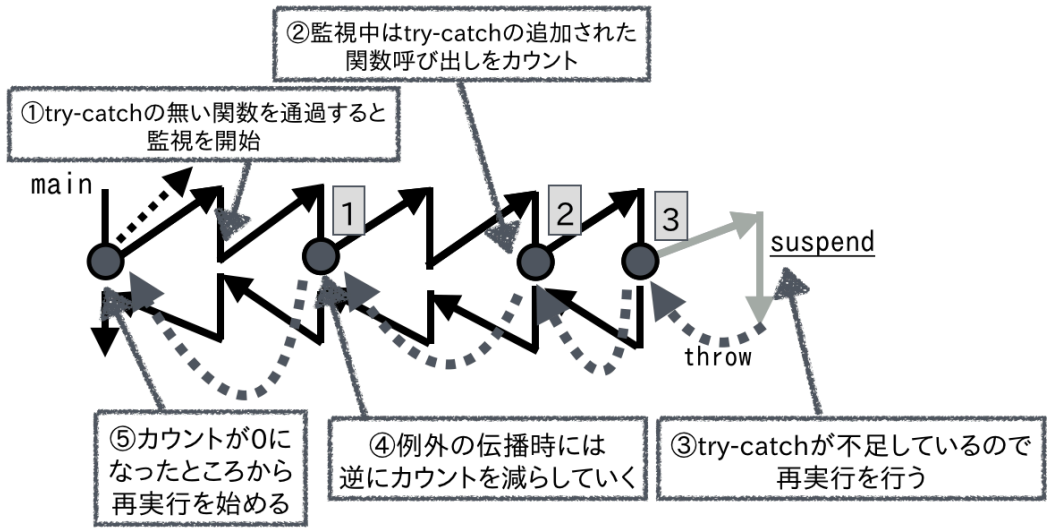

図 5.2: try-catch 挿入の不足を監視する機構

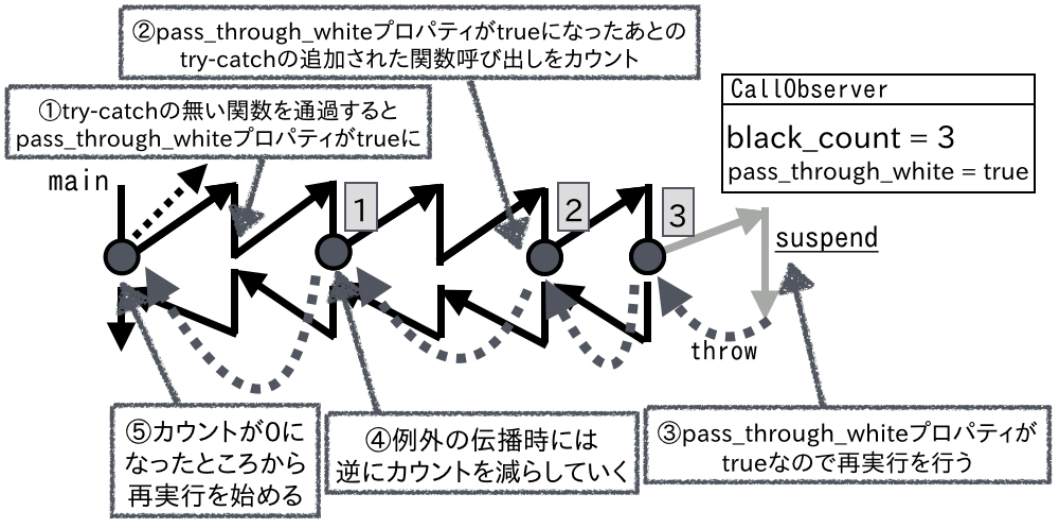

図 5.3: CallObserver の動作

まず、try-catch の挿入されていない関数を通過すると、関数呼び出しの監視を開始する (コード 5.3 の 7 行における CallObserver.passWhite 関数の実行)。監視が開始されてからの try-catch の挿入された関数を追加した数をカウントしていく (コード 5.3 の 18、19 行の命令と、コード 5.2 中の 29、30 行)。もし、継続キャプチャ関

数まで至ること無く、try-catch の挿入された関数から制御が返ってきた場合には 1 つずつカウントを減らしていく (コード 5.3 の 21、22 行の命令と、コード 5.2 中の 32、33 行)。カウントが 0 になり try-catch の無い関数呼び出しにまで返ってきた時点で監視を中止する。もしくは、継続キャプチャ関数が呼び出された場合には、再実行ポイントまで制御を戻すための例外の伝播を開始する。例外を伝播し、catch 節を通過するたびにカウントを減らしていき、カウントが 0 になったポイントから、再実行を開始する。

5.4 プロファイラによる本体の切り換え

関数呼び出し回数を監視し、回数に応じて関数の状態を変更するためのプロファイラをプログラム変換時に追加する。全ての関数は関数宣言時にプロファイラへ自分の参照を登録するようになっており、コード 5.2 において 60 行目の命令でグローバルオブジェクトである Profiler オブジェクトに自身の参照を追加している。コード 5.2 の 3 行目の変数*__call_count* の値は関数呼び出し回数であり、プロファイラはこの変数に定期的にアクセス (コード 5.2 の 37 – 44 行における self.check 関数の実行) し、値を監視する。関数呼び出し回数があらかじめ定められたしきい値を超えると try-catch の削除を行う。図 5.4 のようなコールフローのプログラムが一定回数実行された結果が、表 5.1 だとする。このとき、関数 f2 からは try-catch が削除され、図 5.5 のようにプログラムの高速化が行われる。

表 5.1: 関数が呼び出された回数の例 (しきい値 70 の場合)

関数名	呼び出された回数
main	50
g	50
f1	80
f2	200

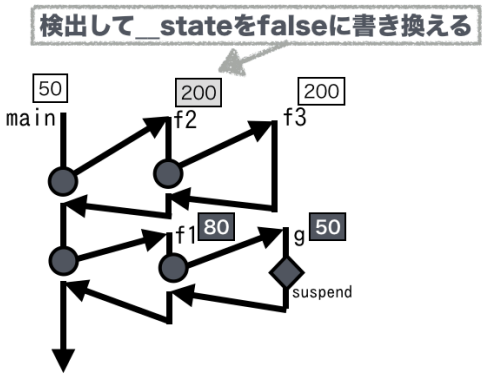

図 5.4: プロファイラの役割

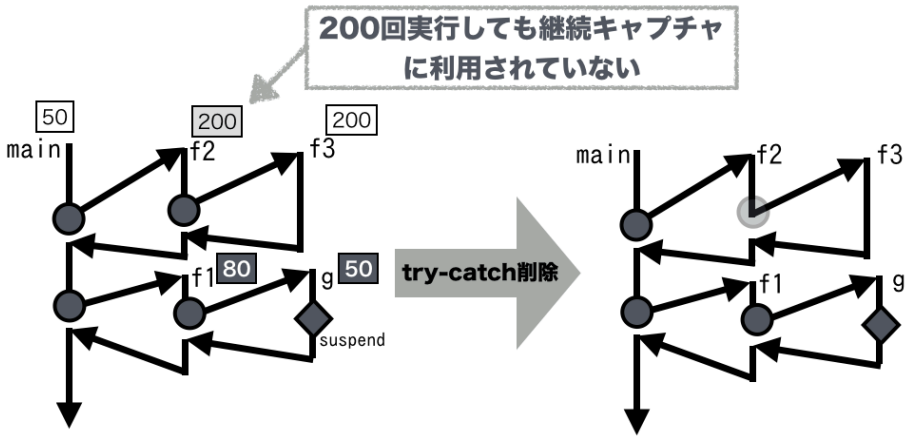

図 5.5: 実行時情報による try-catch 削除

5.5 try-cacth が挿入された関数の本体のコード

コード 5.2 の 12 行目の関数 funcB の内容をコード 5.5 に示す。

ソースコード 5.5: try-catch を含む関数

```
1  var funcB = function f1( x, g ) {
2    var t1, t2;
3
4    var k1 = function() {
5      t1 = "f1: " + x;
6      print(t1);
7      var loop_flg = true;
8      while ( loop_flg ) {
9        loop_flg = false;
10       try {
11         t2 = g();
12       } catch ( ex ) {
```

```
13            if( ex instanceof FailCaptureException &&
14                CallObserver.black_count() > 0 ) {
15              CallObserver.dec();
16              throw ex;
17            }
18            if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
19              FailCapture.flg = true; // 復元フラグを立てる
20              loop_flg = true;
21              CallObserver.reset();
22            }
23            if ( ex instanceof ContinuationException ) {
24              ex.pushFrame(function( t2 ){
25                k2( t2 );
26              });
27              __is_access = true;
28            }
29            if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                  {
30              throw ex;
31            }
32          }
33        }
34        return k2( t2 );
35      };
36      var k2 = function( t2 ) {
37        return t2;
38      };
39      return k1();
40    };
```

コード 5.5 の 7 – 9 行の変数 loop_flg の宣言・初期化と while 文のコードは、大域脱出と再実行時による try-catch 挿入における、再実行時に必要な goto 命令のエミュレートに用いられる。もし、この関数が再実行ポイントとなった場合には変数 loop_flg を true に変更し、関数呼び出しの再実行を行う。

catch 節において、23 – 28 行のコードは継続キャプチャに用いられるものである。この部分には、24 – 26 行は従来手法における残りの計算を保存するコードである。また加えて、27 行目にこの関数が継続キャプチャに利用されたかを判定する変数__is_access への true の代入のコードが記述されている。このコードは継続のキャプチャ時に発生する例外の伝播によって実行される。変数__is_access が true に書き換わることで、対象の関数は図 5.1 における下の状態に遷移し、以降はずっと try-catch が挿入された状態で呼び出される。

13 – 17 行と 18 – 22 行の分岐は大域脱出と再実行に利用されるコードである。13 – 17 行は大域脱出の処理中かを判定し、大域脱出の途中であれば CallObserver.dec 関数を実行し、前の関数呼び出し元に再び例外を throw する処理である。18 – 22

行はこの関数が復元ポイントだった場合の処理となり、復元フラグを true に変更し (FailCapture.flg が true のときの関数呼び出しはすべて try-catch が含まれるものとなる)、CallObserver オブジェクトの状態をリセットし、関数呼び出しの再実行を行う。

第 6 章

性能評価と考察

現在公開されている主要な JavaScript 処理系 (Mozilla SpiderMonkey と Google V8) において、特定ベンチマークにおいて提案手法が有効に動作し、実際にオーバーヘッドを削減し性能向上ができるのかを検証するための性能評価実験を行った。実験は、CPU は Intel Core i5(1.7 GHz)、メモリは 4GB(1333 MHz DDR3)、OS は MacOSX(10.9.1) のマシンにおいて行った。

6.1 Tak 関数による性能評価

シンプルなベンチマークとして、Tak の実行中に suspend することが決してなく try-catch が不要であると想定した場合のベンチマークを測定した。既存手法と提案手法の比較を行っており、既存手法においては、try-catch 挿入による変換後の Tak の実行のオーバーヘッドの測定を行った。また、提案手法においては、動的に try-catch を取り除くしくみを導入する変換をした Tak を用意し、try-catch が動的に取り除かれる前、取り除かれた後のそれぞれのオーバーヘッドの測定を行った。

ソースコード 6.1: Tak 関数

```
1 function tak(x, y, z) {
2   if (x <= y) return z;
3   return tak(tak(x − 1, y, z),
4              tak(y − 1, z, x),
5              tak(z − 1, x, y));
6 }
```

それぞれの関数呼び出し手法に対し、Tak(15, 10, 0) で計測した。try-catch が動的に取り除かれる前、取り除かれた後のそれぞれのオーバーヘッド、これらの 3 種

類のオーバーヘッドを比較した結果を表 6.1 に示す。

この結果から全ての関数呼び出しのうち約 3 割以上が try-catch を含まない関数呼び出しとなったとき、提案手法は既存手法より高速に動作することが予想される。

表 6.1: Tak 関数によるベンチマーク

関数の呼び出し方式	実行時間 (s)	
	SpiderMonkey	v8
既存手法	2.99	0.75
提案手法 (try-catch の削減前)	3.65	0.95
提案手法 (try-catch の削減後)	0.24	0.18

6.2 コルーチンを実現したプログラムの性能評価

次に 2 つの手続き間で制御を渡し合いながら処理を進めるようなコルーチンの制御構造を実現したプログラム 6.2 のベンチマークを測定した。

このプログラムは、2 つの手続き task1 と task2 を間で処理を中断しながら、制御を渡しあう処理を行う (図 6.1 参照)。task1、task2 はともに tak 関数を繰り返し実行するプログラムとなっており、task1 は tak1(14, 8, 0) を、task2 は tak2(10, 5, 0) を繰り返し実行する。task1 は Tak 実行中は suspend することなく、Tak が実行し終わる毎に一度 suspend を行う。task2 は Tak 実行中にある条件を満たしたときに suspend を行う。task2 が 4 回実行されるとプログラムは終了する。

このベンチマークプログラムは、task2 に比べ task1 の実行時間が非常に大きい。task1 は提案手法により高速化 (try-catch の除去) が可能であり、task2 は高速化が不可能である。

ソースコード 6.2: 相互に別の Tak 関数を実行するコルーチンのプログラム

```
1 var Resume1, Resume2;
2 var tak1 = function (x, y, z) {
3   if (x <= y) return z;
4   return tak1(tak1(x − 1, y, z),
```

```
5                 tak1(y − 1, z, x),
6                 tak1(z − 1, x, y));
7  };
8  var tak2 = function (x, y, z) {
9    if (x <= y) return z;
10   if ( x === 7 && y === 1 ) // Tak 再帰中で条件を満たしたときに suspend
11     suspend( function( cont ) {
12       Resume2 = cont;
13       if (!Resume1){
14         topLevel( task1 );
15         return;
16       }
17       Resume1(); // 別のコルーチンに制御を渡す
18     } );
19   return tak2(tak2(x − 1, y, z),
20                 tak2(y − 1, z, x),
21                 tak2(z − 1, x, y));
22 };
23 var task1 = function() {
24   tak1( 14, 8, 0 );
25   suspend( function( cont ){ // task1 の再帰一回毎に suspend
26     Resume1 = cont;
27     if (!Resume2) {
28       topLevel(function(){ task2( 3 ); });
29       return;
30     }
31     Resume2(); // 別のコルーチンに制御を渡す
32   } );
33   task1();
34 };
35 var task2 = function ( i ) {
36   if (i < 0)
37     return;
38   tak2( 10, 5, 0 );
39   print('task2:␣' + i);
40   task2( i − 1 );
41 };
42 topLevel( task1 );
```

測定結果 6.2、プログラム 6.1 の実行に提案手法を適用することで大きな性能向上を図ることができることを確認した。suspend をそれほど頻繁に行わず、実行時間のほとんどを try-catch が削除可能な関数の実行が占めるプログラムの実行においては、提案手法が有効であることを確認した。

表 6.2: コルーチンプログラムによるベンチマーク

関数の呼び出し方式	実行時間 (s)	
	SpiderMonkey	v8
既存手法	110.73	25.8
提案手法 (try-catch の削減前)	124.70	31.28
提案手法 (try-catch の削減後)	7.27	3.82

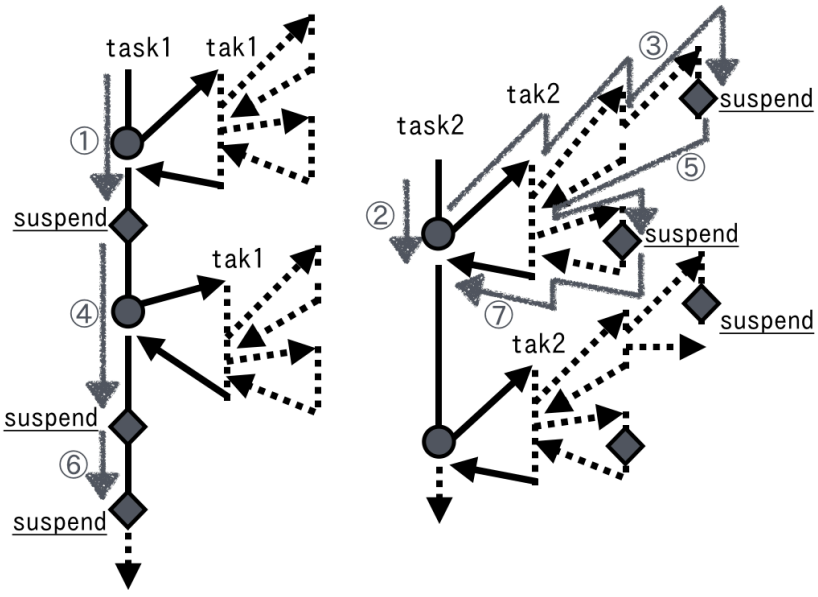

図 6.1: 相互に呼び出し合うコルーチンのプログラムのコールフロー

第 7 章

課題

7.1 静的プログラム解析との併用

本手法は静的プログラム解析技術を併用することで、より高度な高速化を行うことができる。呼び出し先で suspend 関数が呼び出される可能性がない関数呼び出しを safeCall、呼び出し先で suspend 関数が少しでも呼び出される可能性がある関数呼び出しを unsafeCall と呼ぶ。また、unsafeCall を含む関数を unsafeFunction、含まれない関数を safeFunction と呼ぶこととする。

静的プログラム解析を行い、全ての関数宣言、関数呼び出しに関して、unsafeFunction か safeFunction か、unsafeCall か safeCall かを判別するためのコメントを付けたプログラムをコード 7.1 に示す。

ソースコード 7.1: sample プログラムに静的解析を行った例

```
1  var Resume;
2  // unsafeFunction
3  var f1 = function( x, g ) {
4    print('f1:␣' + x); // safeCall
5    return g(); // unsafeCall
6  };
7  // safeFunction
8  var f2 = function( x ) {
9    print('f2:␣' + x); // safeCall
10   return f3( x * 10 ); // safeCall
11 };
12 // safeFunction
13 var f3 = function( x ) {
14   print('f3:␣' + x); // safeCall
15   return x + 1;
16 };
17 // unsafeFunction
18 var g = function () {
19   suspend(function( cont ){ // 継続キャプチャ関数 (unsafeCall)
20     print('suspend');
```

```
21      Resume = cont;
22    });
23  };
24  // unsafeFunction
25  var main = function (){
26    f2( 1 ); // safeCall
27    f1( 2, g ); // unsafeCall
28    f2( 3 ); // safeCall
29  };
```

このプログラムにおいて、関数 f1 に関しては引数に関数を受け取っており、実行しするまで継続のキャプチャに用いられるかどうかの判別が不可能である。

このような関数は全て unsafe と見なされ、継続のキャプチャに備える必要があるが、unsafe な関数においても、全く継続のキャプチャに用いられない場合が多く考えられる。このような場合において、本研究の提案手法を適用し、静的プログラム解析と併用を行うことでさらなる性能向上が見込める。

第 8 章

結論

本研究では、既存手法として知られている例外処理コード (try-catch) を追加するプログラム変換において、例外処理がオーバーヘッドの大きな原因となっている点に着目し、実行時情報を活用した動的な try-catch の追加／削除を行いオーバーヘッドを削減する新しい手法を提案した。

まずは、既存の発表 [4][6][5] を元に try-catch を追加し、継続のキャプチャを実現する方法と実装を示した。次に、既存手法の実装を元に実行時情報を活用した動的な try-catch 挿入／削除を実現するための設計と実装を示し、どのような戦略で try-catch 挿入／削除を行うかを説明した。

提案手法による、動的な try-catch 挿入／削除のプログラム変換では全ての関数に分岐を追加するため、それだけでオーバーヘッドが大きくなる。しかし、追加された分岐のオーバーヘッドに比べ、try-catch 追加によるオーバーヘッドの方が大きいため、ある程度の try-catch の削除を行うことができれば、既存手法よりオーバーヘッドの少ないプログラムの実行が実現できる。

性能評価によって、提案手法導入による分岐オーバーヘッド増加に比べ、try-catch 削減によるオーバーヘッドの削減が十分に大きいことを確認した。プログラム変換によって第一級継続を JavaScript に実現する際に本手法を適用することでオーバーヘッド削減が可能である。

謝辞

本研究は、電気通信大学大学院 情報システム学研究科の情報システム基盤学 基盤ソフトウェア学講座 小宮研究室にて行ったものです。

研究を進めるに際して、関連研究の紹介や研究テーマ、研究の内容の詳細、修士論文執筆に至るまで多くのご指導をいただきました指導教員の小宮常康先生に心から深く感謝致します。研究遂行に必要な指導、助言、励ましをいただきました多田好克先生に大変感謝致します。研究に関係する書籍の輪読会に関して、熱くご指導いただき、また研究の進め方に関しても多くの助言をいただきました荒堀喜貴先生に感謝しております。修士課程で研究を進める上で必要な助言をいただきました佐藤喬先生に感謝致します。

本研究に必要な知識を養うにあたっては、研究論文の輪読会を開催していただくなど、基盤ソフトウェア学講座の方々にも大変有益な議論をしていただきました。あらためて、本研究を進めるに際してお世話になりました基盤ソフトウェア学講座の方々に感謝の意を表します。

参考文献

[1] ケント・ディヴィグ著, 村上雅章訳, プログラミング言語SCHEME, ピアソンエデュケーション, 2000.(ISBN 4-89471-226-1.)

[2] R. Kelsey, W. Clinger, J. Rees (eds.), Revised5 Report on the Algorithmic Language Scheme, Higher-Order and Symbolic Computation, 11(1), September, 1998 and ACM SIGPLAN Notices, 33(9), October, 1998. `http://www.schemers.org/Documents/Standards/R5RS/`.

[3] William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation strategies for first-class continuations. In Journal of Higher Order and Symbolic Computation, 12(1), 1999, pages 7 – 45.

[4] Sekiguchi, T., Sakamoto, T., and Yonezawa, A. Portable implementation of continuation operators in imperative languages by exception handling. Lecture Notes in Computer Science 2022 (2001), pp. 217 – 233.

[5] Florian Loitsch, Exceptional Continuations in JavaScript, Proceedings of the 2007 Workshop on Scheme and Functional Programming, pp. 37 – 46.

[6] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. Con- tinuations from generalized stack inspection. In International Conference on Functional Programming 2005, pp. 216 – 227.

[7] 康　娜丹，小宮常康. プログラム変換器によるメモ化の適用範囲拡張. 電気通信大学大学院情報システム学研究科情報システム基盤学専攻 第 74 回全国大会講演論文集, pp. 431 – 433.

付録A

付録

A.1 ベンチマークプログラム

ソースコード A.1: コルーチンを用いたベンチマークプログラム

```
1  // 定数
2  const FUNC_B = false;
3  const FUNC_A = true;
4
5  const T_FUNCAL_COUNT = 5; // 関数切り替えのしきい値
6
7  // 復元オブジェクト (継続オブジェクトを保存するグローバル変数)
8  var Resume1 = null;
9  var Resume2 = null;
10
11 // ----------------------------------------
12 // 継続オブジェクトの定義
13 function Continuation () {
14   this.frames = [];
15 }
16 Continuation.prototype.pushFrame = function ( f ) {
17   this.frames.push( f );
18 };
19 Continuation.prototype.isEmpty = function () {
20   return this.frames.length === 0;
21 };
22 Continuation.prototype.setFrames = function( frames ) {
23   this.frames = this.frames.concat( frames );
24 };
25
26 // 継続オブジェクトの実行
27 Continuation.prototype.execCont = function () {
28   var temp = null;
29   var cont = this;
30   var f;
31   try {
32     while ( !this.isEmpty() ) {
33
34       f = cont.frames.shift();
35       temp = f( temp );
36
37     }
38     return temp;
```

```
39
40    } catch ( ex ) {
41
42      if ( ex instanceof ContinuationException ) {
43        ex.setFrames( cont.frames );
44
45        // suspend の引数の関数の実行
46        var result = ex.execFunc();
47        return result;
48      }
49      throw ex;
50
51
52    }
53  };
54
55  // 継続例外オブジェクト
56  function ContinuationException () {
57    this.contination = new Continuation();
58  };
59  ContinuationException.prototype.pushFrame = function( frame ) {
60    this.contination.pushFrame( frame );
61  };
62  ContinuationException.prototype.setFrames = function( frames ) {
63    this.contination.setFrames( frames );
64  };
65  ContinuationException.prototype.setFunc = function( func ) {
66    this.func = func;
67  };
68  ContinuationException.prototype.execFunc = function() {
69    var that = this;
70    this.func( function(){
71      that.contination.execCont(); });
72  };
73
74  ContinuationException.prototype.getContinuation = function() {
75    return this.contination;
76  };
77
78  //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
79  // suspend 関数
80  function suspend( func ) {
81
82    if ( CallObserver.isPassWhite() ) {
83      print( 'throw␣new␣FailCaptureException();' );
84      throw new FailCaptureException();
85    }
86
87    FailCapture.flg = false;
88    var cex = new ContinuationException();
89    cex.setFunc( func );
90    throw cex;
91  }
92
93  // −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
94  // Main Routine TopLevel
95  function topLevel( main ) {
96    var loop_flg = true;
97
98    while ( loop_flg ) {
```

```
99      loop_flg = false;
100     try {
101
102       print('top');
103       main();
104       return;
105
106     } catch ( ex ) {
107
108       // 復元
109       if ( ex instanceof FailCaptureException ) {
110         CallObserver.reset();
111         print('re-exec');
112         FailCapture.flg = true; // 復元フラグを立てる
113         loop_flg = true;
114       }
115
116       // 継続キャプチャ
117       if ( ex instanceof ContinuationException ) {
118         // suspend の引数の関数の実行
119         ex.execFunc();
120         return;
121       }
122       if ( !( ex instanceof FailCaptureException) ) {
123         throw ex;
124       }
125     }
126   }
127 };
128
129 // 関数の解析部
130 var Profiler = (function() {
131
132   // 現在のどこまで監視したかのindex
133   var seek_index = 0;
134   // var check_func_count = 0;
135
136   // 監視対象の関数群
137   var watch_funcs = [];
138
139   // 全ての関数の連想配列
140   var all_funcs = {};
141
142   var switchToHighSpeed = function( index ) {
143     var func = watch_funcs.splice(index, 1);
144     var func_name = func.fname;
145     func.switchSpeed( FUNC_A );
146   };
147
148   var switchToLowSpeed = function( func_name ){
149     var func = all_funcs[ func_name ];
150
151     func.switchSpeed( FUNC_B );
152   };
153
154   // accessor
155   return {
156     // regist
157     registFunction: function ( func ) {
158       all_funcs[func.fname] = func;
```

```
159         watch_funcs.push( func );
160       },
161
162       // check
163       checkFunctions: function () {
164         var length = watch_funcs.length;
165         while (seek_index < length) {
166           print(watch_funcs[ seek_index ].fname);
167           watch_funcs[ seek_index ].check();
168           ++seek_index;
169         }
170         seek_index = 0;
171       },
172
173       init: function() {
174         for( key in all_funcs ) {
175           watch_funcs.push( all_funcs[key] );
176         }
177       },
178
179       swichAllHigh: function() {
180
181         for ( i in watch_funcs ){
182           // print(watch_funcs[i].accessState());
183           watch_funcs[i].switchSpeed( FUNC_A );
184           // print(watch_funcs[i].accessState());
185         }
186
187       }, swichAllLow: function() {
188         for ( i in watch_funcs ){
189           watch_funcs[i].switchSpeed( FUNC_B );
190         }
191       }, infoFunctions: function() {
192         for ( name in all_funcs ) {
193           var obj = all_funcs[name].info();
194           print('name:␣' + name + '\n' +
195                 'state:␣' + obj.state + '\n' +
196                 'call_count:␣' + obj.call_count + '\n' +
197                 'is_access:␣' + obj.is_access);
198           print('---');
199         }
200       }
201
202     };
203   })();
204
205   // ------------------------
206   // 復元例外
207   function FailCaptureException() {}
208
209   // ------------------------
210   // 復元フラグ
211   var FailCapture = new Object();
212   // FailCapture.setFuncFlg = false;
213   FailCapture.flg = false;
214
215   var CallObserver = (function() {
216     var black_count = 0;
217     var pass_white = false;
218
```

```
219   return {
220     plus: function(){
221       ++black_count;
222     },
223     minus: function(){
224       −−black_count;
225       if ( black_count === 0 )
226         pass_white = false;
227
228       if ( black_count < 0) {
229         throw 'black exception';
230       }
231     },
232     passWhite: function(){
233       pass_white = true;
234     },
235     reset: function(){
236       pass_white = false;
237       black_count = 0;
238     },
239     isPassWhite: function(){
240       return pass_white;
241     },
242     black_count: function() {
243       return black_count;
244     }
245   };
246 })();
247
248
249 // 本体 −−−−−−−−−−−−−−−−−−−−−−−−
250 // −−−−−−−−−−−−−−−−
251 var tak1 = (function(){
252   // status
253   var __state = FUNC_B;
254   var __call_count = 0;
255   var __is_access = false;
256
257   var funcA = function (x, y, z) {
258     if (x <= y) return z;
259     return tak1(tak1(x − 1, y, z),
260                 tak1(y − 1, z, x),
261                 tak1(z − 1, x, y));
262   };
263
264   var funcB = function (x, y, z) {
265     var t1, t2, t3, t4, t5, t6, t7, t8;
266     var k1 = function() {
267       t1 = x <= y;
268       if (t1)
269         return z;
270       t2 = x − 1.0;
271       var loop_flg = true;
272       while ( loop_flg ) {
273         loop_flg = false;
274         try {
275           t3 = tak1(t2, y, z);
276         } catch ( ex ) {
277           if( ex instanceof FailCaptureException &&
278               CallObserver.black_count() > 0 ) {
```

```
279             CallObserver.minus();
280             __is_access = true;
281           throw ex;
282           }
283           if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
284             FailCapture.flg = true; // 復元フラグを立てる
285             __is_access = true;
286             loop_flg = true;
287             CallObserver.reset();
288           }
289           if ( ex instanceof ContinuationException ) {
290             __is_access = true;
291             ex.pushFrame(function( t3 ) {
292               return k2( t3 );
293             });
294           }
295           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                {
296             throw ex;
297           }
298         }
299       }
300       return k2( t3 );
301     };
302
303     var k2 = function ( t3 ) {
304       t4 = y - 1.0;
305       var loop_flg = true;
306       while ( loop_flg ) {
307         loop_flg = false;
308         try {
309           t5 = tak1(t4, z, x);
310         } catch ( ex ) {
311           if( ex instanceof FailCaptureException &&
312               CallObserver.black_count() > 0 ) {
313             CallObserver.minus();
314             __is_access = true;
315             throw ex;
316           }
317           if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
318             FailCapture.flg = true; // 復元フラグを立てる
319             loop_flg = true;
320             __is_access = true;
321             CallObserver.reset();
322           }
323           if ( ex instanceof ContinuationException ) {
324             __is_access = true;
325             ex.pushFrame(function( t5 ){
326               return k3( t3, t5 );
327             });
328           }
329           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                {
330             throw ex;
331           }
332         }
333       }
334       return k3( t3, t5 );
335     };
336
```

```
337     var k3 = function( t3, t5 ) {
338       t6 = z − 1.0;
339       var loop_flg = true;
340       while ( loop_flg ) {
341         loop_flg = false;
342         try {
343           t7 = tak1(t6, x, y);
344         } catch ( ex ) {
345           if( ex instanceof FailCaptureException &&
346                 CallObserver.black_count() > 0 ) {
347             CallObserver.minus();
348             __is_access = true;
349             throw ex;
350           }
351           if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
352             FailCapture.flg = true; // 復元フラグを立てる
353             loop_flg = true;
354             __is_access = true;
355           CallObserver.reset();
356           }
357           if ( ex instanceof ContinuationException ) {
358             __is_access = true;
359             ex.pushFrame(function( t7 ){
360               return k4( t3, t5, t7 );
361             });
362           }
363           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                  {
364             throw ex;
365           }
366         }
367       }
368       return k4( t3, t5, t7 );
369     };
370
371     var k4 = function( t3, t5, t7 ) {
372       var loop_flg = true;
373       while ( loop_flg ) {
374         loop_flg = false;
375         try {
376           t8 = tak1( t3, t5, t7 );
377         } catch ( ex ) {
378           if( ex instanceof FailCaptureException &&
379             CallObserver.black_count() > 0 ) {
380             CallObserver.minus();
381             __is_access = true;
382             throw ex;
383           }
384
385           if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
386             FailCapture.flg = true; // 復元フラグを立てる
387             loop_flg = true;
388             __is_access = true;
389             CallObserver.reset();
390           }
391
392           if ( ex instanceof ContinuationException ) {
393             __is_access = true;
394             ex.pushFrame(function( t8 ){
395               return t8;
```

```
396             });
397           }
398           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                  {
399             throw ex;
400           }
401         }
402       }
403       return t8;
404     };
405     return k1();
406   };
407
408   // return function
409   var self = function( x, y, z ) {
410     var result;
411     // キャプチャ失敗時の再実行
412     if (FailCapture.flg || __is_access) {
413       // print( FailCapture.flg, self.fname , __is_access);
414       // __is_access = true;
415       ++__call_count;
416       __state = FUNC_B;
417       result = funcB( x, y, z );
418       return result;
419     }
420     return self.body( x, y, z );
421   };
422   self.body = function( x, y, z ){
423     var result;
424     ++__call_count;
425     // FUNC_B function
426     if ( CallObserver.isPassWhite() )
427       CallObserver.plus();
428     result = funcB( x, y, z );
429     if ( CallObserver.isPassWhite() )
430       CallObserver.minus();
431     return result;
432   };
433   // check
434   self.check = function(){
435     if (__call_count > T_FUNCAL_COUNT &&
436         !__is_access) {
437       self.switchSpeed(FUNC_A);
438       return true;
439     }
440     return false;
441   };
442   // switch
443   self.switchSpeed = function ( state ) {
444     if ( state ) {
445       __state = FUNC_A;
446       print('high');
447       self.body = function( x, y, z ){
448         ++__call_count;
449         return funcA(x, y, z);
450       };
451     } else {
452       __state = FUNC_B;
453       self.body = function( x, y, z ) {
454         var result;
```

```
455           ++__call_count;
456           // FUNC_B function
457           if ( CallObserver.isPassWhite() )
458             CallObserver.plus();
459           result = funcB( x, y, z );
460           if ( CallObserver.isPassWhite() )
461             CallObserver.minus();
462           return result;
463         };
464       }
465     };
466     self.info = function() {
467       return {
468         state: __state,
469         call_count: __call_count,
470         is_access: __is_access
471       };
472     };
473     self.fname = "tak1";
474     self.accessState = function(){
475       return __state;
476     };
477     Profiler.registFunction( self );
478     return self;
479   })();
480
481   var tak2 = (function(){
482     // status
483     var __state = FUNC_B;
484     var __call_count = 0;
485     var __is_access = false;
486
487     var funcA = function ( x, y, z ) {
488       if (x <= y) return z;
489       if ( x === 7 && y === 1 ) // Tak 再帰中で条件を満たしたときに suspend
490         suspend( function( cont ) {
491           Resume2 = cont;
492           if (!Resume1){
493             topLevel( taskLoop1 );
494             return;
495           }
496           Resume1(); // 別のコルーチンを実行する
497         } );
498       return tak2(tak2(x − 1, y, z),
499                   tak2(y − 1, z, x),
500                   tak2(z − 1, x, y));
501     };
502
503     var funcB = function (x, y, z) {
504       var t1, t2, t3, t4, t5, t6, t7, t8;
505
506       var k1 = function() {
507         t1 = x <= y;
508         if (t1)
509           return z;
510         var loop_flg = true;
511         while ( loop_flg ) {
512           loop_flg = false;
513           try {
514             if ( x === 7 && y === 1 ) //
```

```
515             return suspend( function( cont ){
516               // print('suspend2');
517               Resume2 = cont;
518               if (Resume1===null)
519                 topLevel( taskLoop1 );
520               if (Resume1!==null)
521                 Resume1();
522             } );
523           } catch ( ex ) {
524
525             if( ex instanceof FailCaptureException &&
526                 CallObserver.black_count() > 0 ) {
527               CallObserver.minus();
528               __is_access = true;
529               throw ex;
530             }
531             if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
532               FailCapture.flg = true; // 復元フラグを立てる
533               loop_flg = true;
534               __is_access = true;
535               CallObserver.reset();
536             }
537             if ( ex instanceof ContinuationException ) {
538               __is_access = true;
539               ex.pushFrame(function() {
540                 return k2();
541               });
542             }
543             if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                    {
544               throw ex;
545             }
546           }
547         }
548         return k2();
549       };
550       var k2 = function(){
551         t2 = x - 1.0;
552         var loop_flg = true;
553         while ( loop_flg ) {
554           loop_flg = false;
555           try {
556             t3 = tak2( t2, y, z );
557           } catch ( ex ) {
558             if( ex instanceof FailCaptureException &&
559                 CallObserver.black_count() > 0 ) {
560               CallObserver.minus();
561               __is_access = true;
562               throw ex;
563             }
564             if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
565               FailCapture.flg = true; // 復元フラグを立てる
566               loop_flg = true;
567               __is_access = true;
568               CallObserver.reset();
569             }
570             if ( ex instanceof ContinuationException ) {
571               __is_access = true;
572               ex.pushFrame(function( t3 ) {
573                 return k3( t3 );
```

```
574             });
575           }
576           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                  {
577             throw ex;
578           }
579         }
580       }
581       return k3( t3 );
582     };
583
584     var k3 = function( t3 ) {
585       t4 = y - 1.0;
586       var loop_flg = true;
587       while ( loop_flg ) {
588         loop_flg = false;
589         try {
590           t5 = tak2( t4, z, x );
591         } catch ( ex ) {
592           if( ex instanceof FailCaptureException &&
593                 CallObserver.black_count() > 0 ) {
594             CallObserver.minus();
595             __is_access = true;
596             throw ex;
597           }
598           if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
599             FailCapture.flg = true; // 復元フラグを立てる
600             loop_flg = true;
601             __is_access = true;
602             CallObserver.reset();
603           }
604           if ( ex instanceof ContinuationException ) {
605             __is_access = true;
606             ex.pushFrame(function( t5 ) {
607               return k4( t3, t5 );
608             });
609           }
610           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                  {
611             throw ex;
612           }
613         }
614       }
615       return k4( t3, t5 );
616     };
617     var k4 = function( t3, t5 ) {
618       t6 = z - 1.0;
619       var loop_flg = true;
620       while ( loop_flg ) {
621         loop_flg = false;
622         try {
623           t7 = tak2( t6, x, y );
624         } catch ( ex ) {
625           if( ex instanceof FailCaptureException &&
626                 CallObserver.black_count() > 0 ) {
627             CallObserver.minus();
628             __is_access = true;
629           throw ex;
630           }
631           if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
```

```
632             FailCapture.flg = true; // 復元フラグを立てる
633             loop_flg = true;
634             __is_access = true;
635             CallObserver.reset();
636           }
637           if ( ex instanceof ContinuationException ) {
638             __is_access = true;
639             ex.pushFrame(function( t7 ) {
640               return k5( t3, t5, t7 );
641             });
642           }
643           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                {
644             throw ex;
645           }
646         }
647       }
648       return k5( t3, t5, t7 );
649     };
650     var k5 = function( t3, t5, t7 ) {
651       var loop_flg = true;
652       while ( loop_flg ) {
653         loop_flg = false;
654         try {
655           t8 = tak2(t3, t5, t7);
656         } catch ( ex ) {
657           if( ex instanceof FailCaptureException &&
658               CallObserver.black_count() > 0 ) {
659             CallObserver.minus();
660             __is_access = true;
661             throw ex;
662           }
663           if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
664             FailCapture.flg = true; // 復元フラグを立てる
665             loop_flg = true;
666             __is_access = true;
667             CallObserver.reset();
668           }
669           if ( ex instanceof ContinuationException ) {
670             __is_access = true;
671             ex.pushFrame(function( t8 ) {
672               return k6( t3, t5, t7, t8 );
673             });
674           }
675           if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                {
676             throw ex;
677           }
678         }
679       }
680       return k6( t3, t5, t7, t8 );
681     };
682     var k6 = function( t3, t5, t7, t8 ){
683       return t8;
684     };
685     return k1();
686   };
687
688   // return function
689   var self = function( x, y, z ) {
```

```
690     var result;
691     // キャプチャ失敗時の再実行
692     if (FailCapture.flg || __is_access) {
693       __is_access = true;
694       ++__call_count;
695       __state = FUNC_B;
696       result = funcB( x, y, z );
697       return result;
698     }
699     return self.body(x, y, z);
700   };
701   self.body = function( x, y, z ){
702     var result;
703     ++__call_count;
704     // FUNC_B function
705     if ( CallObserver.isPassWhite() )
706       CallObserver.plus();
707     result = funcB( x, y, z );
708     if ( CallObserver.isPassWhite() )
709       CallObserver.minus();
710     return result;
711   };
712   // check
713   self.check = function(){
714     if (__call_count > T_FUNCAL_COUNT &&
715         !__is_access) {
716       self.switchSpeed(FUNC_A);
717       return true;
718     }
719     return false;
720   };
721   // switch
722   self.switchSpeed = function ( state ) {
723     if ( state ) {
724       __state = FUNC_A;
725       self.body = function( x, y, z ) {
726         var result;
727         ++__call_count;
728         CallObserver.passWhite();
729         result = funcA( x, y, z );
730         if ( CallObserver.black_count() === 0 )
731           CallObserver.reset();
732         return result;
733       };
734     } else {
735       __state = FUNC_B;
736       self.body = function( x, y, z ) {
737         var result;
738         ++__call_count;
739         // FUNC_B function
740         if ( CallObserver.isPassWhite() )
741           CallObserver.plus();
742         result = funcB( x, y, z );
743         if ( CallObserver.isPassWhite() )
744           CallObserver.minus();
745         return result;
746       };
747     }
748   };
749   self.info = function() {
```

```
750       return {
751         state: __state,
752         call_count: __call_count,
753         is_access: __is_access
754       };
755     };
756     self.fname = "tak2";
757     self.accessState = function(){
758       return __state;
759     };
760     Profiler.registFunction( self );
761     return self;
762   })();
763
764   var taskLoop1 = (function() {
765
766     // status
767     var __state = FUNC_B;
768     var __call_count = 0;
769     var __is_access = false;
770
771     var funcA = function(){
772       print('tak1');
773       tak1( 14, 8, 0 );
774       // print('tak1');
775       suspend( function( cont ) {
776         Resume1 = cont;
777         if (!Resume2) {
778           topLevel(function(){ taskLoop2( 3 ); });
779           return;
780         }
781         Resume2();
782       } );
783       taskLoop1();
784     };
785
786     var funcB = function() {
787       var k1 = function(){
788         var loop_flg = true;
789         while ( loop_flg ) {
790           loop_flg = false;
791           try {
792             // print('tak1');
793             tak1( 14, 8, 0 );
794           } catch ( ex ) {
795             if( ex instanceof FailCaptureException &&
796                 CallObserver.black_count() > 0 ) {
797               CallObserver.minus();
798               __is_access = true;
799               throw ex;
800             }
801             if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
802               FailCapture.flg = true; // 復元フラグを立てる
803               loop_flg = true;
804               __is_access = true;
805               CallObserver.reset();
806             }
807             if ( ex instanceof ContinuationException ) {
808               __is_access = true;
809               ex.pushFrame(function() {
```

```
810                 return k2();
811               });
812             }
813             if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                    {
814               throw ex;
815             }
816           }
817         }
818         return k2();
819       };
820
821       var k2 = function() {
822         var loop_flg = true;
823         while ( loop_flg ) {
824           loop_flg = false;
825           try {
826             suspend( function( cont ){
827               Resume1 = cont;
828               if (Resume2 ===null)
829                 topLevel(function(){ taskLoop2( 3 ); });
830               if (Resume2 !==null)
831                 Resume2();
832             } );
833           } catch ( ex ) {
834             if( ex instanceof FailCaptureException &&
835                 CallObserver.black_count() > 0 ) {
836               __is_access = true;
837               CallObserver.minus();
838             throw ex;
839             }
840             if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
841               FailCapture.flg = true; // 復元フラグを立てる
842               loop_flg = true;
843               __is_access = true;
844               CallObserver.reset();
845             }
846             if ( ex instanceof ContinuationException ) {
847               __is_access = true;
848               ex.pushFrame(function() {
849                 return k3();
850               });
851             }
852             if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                    {
853               throw ex;
854             }
855           }
856         }
857         return k3();
858       };
859       var k3 = function(){
860         taskLoop1();
861       };
862       return k1();
863     };
864     // return function
865     var self = function() {
866       var result;
867       // print('fail');
```

```
868
869       // キャプチャ失敗時の再実行
870       if ( FailCapture.flg || __is_access) {
871         __is_access = true;
872         ++__call_count;
873         __state = FUNC_B;
874         result = funcB();
875         return result;
876       }
877       return self.body();
878     };
879     self.body = function(){
880       var result;
881       ++__call_count;
882       // FUNC_B function
883       if ( CallObserver.isPassWhite() )
884         CallObserver.plus();
885       result = funcB();
886       if ( CallObserver.isPassWhite() )
887         CallObserver.minus();
888       return result;
889     };
890     // check
891     self.check = function(){
892       if (__call_count > T_FUNCAL_COUNT &&
893           !__is_access) {
894         self.switchSpeed(FUNC_A);
895         return true;
896       }
897       return false;
898     };
899
900     // switch
901     self.switchSpeed = function ( state ) {
902       if ( state ) {
903         __state = FUNC_A;
904         print('high');
905         self.body = function() {
906           var result;
907           ++__call_count;
908           CallObserver.passWhite();
909           result = funcA();
910           if ( CallObserver.black_count() === 0 )
911             CallObserver.reset();
912           return result;
913         };
914       } else {
915         __state = FUNC_B;
916         self.body = function( ) {
917           var result;
918           ++__call_count;
919           // FUNC_B function
920           if ( CallObserver.isPassWhite() )
921             CallObserver.plus();
922           result = funcB();
923           if ( CallObserver.isPassWhite() )
924             CallObserver.minus();
925           return result;
926         };
927       }
```

```
928     };
929     self.info = function() {
930       return {
931         state: __state,
932         call_count: __call_count,
933         is_access: __is_access
934       };
935     };
936     self.fname = "taskLoop1";
937     self.accessState = function(){
938       return __state;
939     };
940     Profiler.registFunction( self );
941     return self;
942   })();
943
944   var taskLoop2 = (function(){
945
946     // status
947     var __state = FUNC_B;
948     var __call_count = 0;
949     var __is_access = false;
950
951     var funcA = function ( i ) {
952       if (i < 0)
953         return;
954       tak2( 10, 5, 0 );
955       print('task2:␣' + i);
956       task2( i − 1 );
957     };
958
959     var funcB = function ( i ) {
960
961       var k1 = function() {
962         if (i < 0)
963           return;
964         var loop_flg = true;
965         while ( loop_flg ) {
966           loop_flg = false;
967           try {
968             tak2( 10, 5, 0 );
969           } catch ( ex ) {
970             if( ex instanceof FailCaptureException &&
971                 CallObserver.black_count() > 0 ) {
972               CallObserver.minus();
973               __is_access = true;
974               throw ex;
975             }
976             if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
977               FailCapture.flg = true; // 復元フラグを立てる
978               loop_flg = true;
979               CallObserver.reset();
980               __is_access = true;
981             }
982             if ( ex instanceof ContinuationException ) {
983               __is_access = true;
984               ex.pushFrame(function() {
985                 k2();
986               });
987             }
```

```
988             if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                    {
989               throw ex;
990             }
991           }
992         }
993         k2();
994       };
995
996       var k2 = function(){
997         var loop_flg = true;
998         while ( loop_flg ) {
999           loop_flg = false;
1000          try {
1001            print( 'taskLoop2␣' + i );
1002            taskLoop2( i − 1 );
1003          } catch ( ex ) {
1004            if( ex instanceof FailCaptureException &&
1005                CallObserver.black_count() > 0 ) {
1006              CallObserver.minus();
1007              __is_access = true;
1008            throw ex;
1009            }
1010            if( ex instanceof FailCaptureException && CallObserver.isPassWhite() ) {
1011              FailCapture.flg = true; // 復元フラグを立てる
1012              loop_flg = true;
1013              __is_access = true;
1014              CallObserver.reset();
1015            }
1016            if ( ex instanceof ContinuationException ) {
1017              __is_access = true;
1018            }
1019            if( !(ex instanceof FailCaptureException && CallObserver.isPassWhite() ) )
                    {
1020              throw ex;
1021            }
1022          }
1023        }
1024      };
1025      return k1();
1026    };
1027    // return function
1028    var self = function( i ) {
1029      var result;
1030      // キャプチャ失敗時の再実行
1031      if (FailCapture.flg || __is_access) {
1032        ++__call_count;
1033        __state = FUNC_B;
1034        result = funcB( i );
1035        return result;
1036      }
1037      return self.body( i );
1038    };
1039    self.body = function( i ){
1040      var result;
1041      ++__call_count;
1042      // FUNC_B function
1043      if ( CallObserver.isPassWhite() )
1044        CallObserver.plus();
1045      result = funcB( i );
```

```
1046      if ( CallObserver.isPassWhite() )
1047        CallObserver.minus();
1048      return result;
1049    };
1050    // check
1051    self.check = function(){
1052      if (__call_count > T_FUNCAL_COUNT &&
1053          !__is_access) {
1054        self.switchSpeed(FUNC_A);
1055        return true;
1056      }
1057      return false;
1058    };
1059    // switch
1060    self.switchSpeed = function ( state ) {
1061      print('switchSpeed');
1062      print(state);
1063      if ( state ) {
1064        __state = FUNC_A;
1065        print('high');
1066        self.body = function( i ) {
1067          var result;
1068          ++__call_count;
1069          CallObserver.passWhite();
1070          result = funcA( i );
1071          if ( CallObserver.black_count() === 0 )
1072            CallObserver.reset();
1073          return result;
1074        };
1075      } else {
1076        __state = FUNC_B;
1077        self.body = function( i ) {
1078          var result;
1079          ++__call_count;
1080          // FUNC_B function
1081          if ( CallObserver.isPassWhite() )
1082            CallObserver.plus();
1083          result = funcB( i );
1084          if ( CallObserver.isPassWhite() )
1085            CallObserver.minus();
1086          return result;
1087        };
1088      }
1089    };
1090    self.info = function() {
1091      return {
1092        state: __state,
1093        call_count: __call_count,
1094        is_access: __is_access
1095      };
1096    };
1097    self.fname = "taskLoop2";
1098    self.accessState = function(){
1099      return __state;
1100    };
1101    Profiler.registFunction( self );
1102    return self;
1103  })();
1104
1105  // 提案手法を適用する場合は下記のコメントを外す
```

```
1106 Profiler.swichAllHigh();
1107 Profiler.infoFunctions();
1108 topLevel(taskLoop1);
```